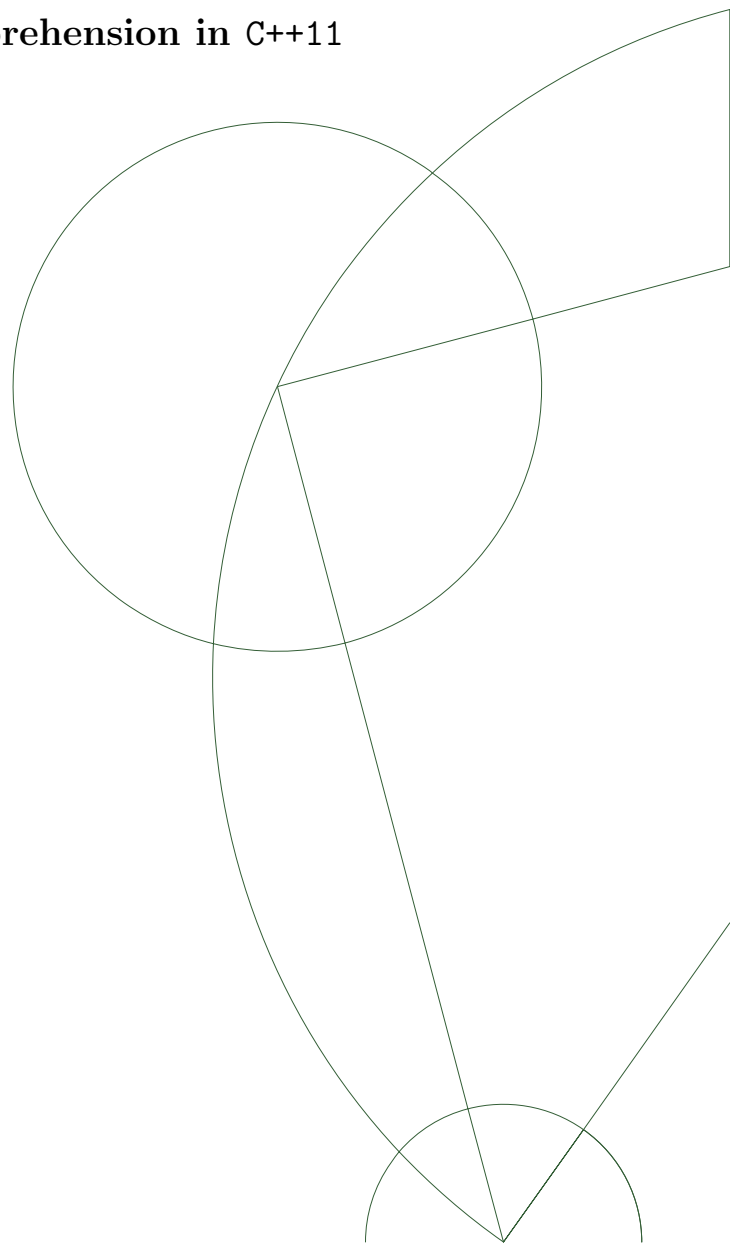
FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Bachelor of Science Project

Martin Holm Cservenka - djp595@alumni.ku.dk
Tue Haulund - qvr916@alumni.ku.dk

Implementing LINQ-like list-comprehension in C++11 using variadic templates



Supervisor: Jyrki Katajainen & Fritz Henglein
9th of June, 2014

Contents

1 Abstract

In the report, the process of constructing a variadic template, header-only library for `C++` implementing `LINQ`-like list-comprehension is explored. A mathematical model of the concept of list-comprehension is presented, which forms the basis for the implementation.

The properties of the variadic template approach is thoroughly compared to existing list-comprehension libraries in a number of aspects, including extensibility, generated binaries, source code size and complexity, type safety and interface usability. The comparisons show that the use of variadic templates as the primary implementation strategy yields a product which is both easily extensible, generates smaller or equally sized binaries and offers compile time type checking. The size of the implementations itself proves to be no longer than existing, equivalent libraries, though the complexity of the code is considerably higher.

In conclusion, the variadic template approach is proven to be both feasible and advantageous in a number of aspect, though not unconditionally so.

2 Introduction

List-comprehension is a powerful, expressive syntactic construct for generating and transforming sequences of various elements. It is a well-known feature in most functional programming languages (`Haskell, LISP`) and in some imperative languages (`JavaScript, Python`).
Using list-comprehension, we can transform a list of elements by applying a chain of functions and operators to the list, which allows for complex transformations on the input in a succinct manner.

Microsofts `.NET` framework includes a similar feature known as `LINQ` (Language Integrated Query). `LINQ` is a more generalized form of list-comprehension, and includes operators which are common to database query languages such as `orderby` and `groupby`.

Although `C++` is widely used [1], the language does not contain a feature similar to list-comprehension or `LINQ`. With the release of the most recent version of the `C++` standard, `C++11`, several additions were made to the core language. One notable addition is variadic templates, which lets a template take a variable number of template parameters [2], allowing for the definition of type-safe variadic functions.

The aim of this project is to explore the possibilities of implementing `LINQ`-like list-comprehension in `C++11` using variadic templates and investigating the advantages in terms of implementation strategy, interface and ease of use.

In order to do so, we will start by presenting a mathematical model of the notion of list-comprehension, the purpose of which is to formalize the theoretical basis and limit the scope of an implementation in a meaningful way. We will then present an experimental implementation of a list-comprehension `C++` library using variadic templates named `stplib` (Sequence Transformation Pipeline library) based on this model.

`stplib` can be found on its GitHub page: `https://github.com/TueHaulund/stplib`

Finally, a presentation of some selected list-comprehension libraries for `C++` is included, along with a brief summary of advantages and disadvantages of `stplib` compared to these libraries, in terms of implementation and interface usability.

2.1 Project Scope and Limitations

Is list-comprehension a suitable application for `C++11` variadic templates, and if so, what are the advantages in terms of implementation, interface, and ease of use?

For the purposes of usefulness and usability, most implementations of list-comprehension or `LINQ` [3] includes a wide range of features and operations. As such features can generally be considered extraneous in the context of the project problem definition, only a select few will be included in the implementation presented in Section 4. See Section 3 for a specific list of basic operations included in the model and implementation.

Similarly, program performance concerns is considered to be beyond the scope of the project. It is thus emphasized that the presented implementation is of an experimental nature and is not suitable for any performance critical purposes.

To ensure a high degree of functional correctness, a reasonably exhaustive collection of tests are included with `stplib`. See Appendix B for instructions on how to build and run the test suite.

3 Mathematical Model

Before the details of a fully functional implementation are explored, a mathematical model of the notion of list-comprehension is detailed.

3.1 Notation and Definitions

When defining the model, we will primarily be working with *sequences* of values. A sequence is similar to a mathematical set, with a few crucial differences. The classic definition of a mathematical set is a collection of unordered and distinct elements [4]. A sequence has a specific ordering, and allows identical elements to appear multiple times. Thus, the sequence $[1, 2, 3, 1]$ is different from the sequence $[1, 1, 2, 3]$, yet both are valid *sequences* of integers, while neither of them would be valid *sets*.

The notion of a sequence is very similar to arrays and other simple data structures from the field of computer programming, which allows for an easier transition when implementing the model in an actual system.

All elements of a sequence must be of the same type. We will refer to the type of the elements of a sequence X as $T(X)$.

$$X = [1.2, -4.5, 3.4] \Rightarrow T(X) = \mathbb{R}$$
$$Y = [\text{"}ab\text{"}, \text{"}ba\text{"}, \text{"}cc\text{"}] \Rightarrow T(Y) = \Sigma^*$$

Where $\mathbb{R}$ is the set of all real numbers and Σ is the set of legal characters in the alphabet for the strings in Y. In this context, the type of a sequence is a *set* of valid values, that each element in the sequence may hold. Most of the operations presented in section 3.2 are only defined for sequences with types for which certain operations are well defined.

An example of this is the $SUM(X)$ operation (section 3.2.6). If $T(X)$ is a set of real numbers or integers, addition is well defined, while if $T(X) = \{1, 2, \text{"}a\text{"}, \text{"}b\text{"}\}$ then $SUM(X)$ is impossible to compute before determining what the result of $1 + \text{"}a\text{"}$ should be.

For the purpose of simplicity we will use set-builder notation to define sequences. Usually this notation is used to specify the constraints on some set of elements, but in this case we will allow duplicate elements and a specific ordering, and thus the result will be a sequence rather than a set.

For the description of the model, various standard mathematical notation will be used:

- $\forall_{0 \leq i \leq n}$ denotes all possible values of i between 0 and n inclusively.
- The functions $MIN(x, y)$ and $MAX(x, y)$ outputs the numerically smallest and largest of their respective arguments.
- If X is a sequence then x_n denotes the nth element of that sequence.
- $|X|$ denotes the number of elements in the sequence X.

If f is some function, we will refer to the domain of f (The set of input values for which it is defined) as $D(f)$, and the codomain (The set of values that f maps the input to) as $C(f)$.

In the following sections, all variables denoted with capitalized letters are sequences, while lowercase variables are either singleton values or functions.

We will refer to any of the functions that alter a sequence as an *operation*, while a composite-function which consists of a series of operations will be referred to as a *transformation*.

3.2 Model Operations

In the following section we will provide definitions for each of the operations supported by the model. The operations are divided into several categories.

3.2.1 Generators

Generally, all operations map some input sequence to an altered output sequence. The generator operations are unique in this regard, since none of them are defined for sequences as input, yet all of them outputs some sequence according to the given arguments.

$$RANGE(n, e, s) = \Big[n, n+s, n+2s, ..., n+xs \;\Big|\; (n+xs \leq e) \wedge (n+((x+1)\cdot s) > e)\Big] \tag{1}$$

$RANGE(n, e, s)$ evaluates to a sequence of numbers ranging from n to at most e using steps of size s. Thus $RANGE(5, 10, 2)$ would result in the sequence $[5, 7, 9]$.

$$EMPTY = \emptyset \tag{2}$$

$EMPTY$ corresponds to the empty sequence $\emptyset$ and is the only nullary operation in the model.

$$SINGLETON(x) = [x] \tag{3}$$

$SINGLETON(x)$ represents the sequence $[x]$.

3.2.2 Filters

These filter operations filters unwanted elements from a sequence.

$$WHERE(X, f) = \Big[x \;\Big|\; x \in X \wedge f(x)\Big] \tag{4}$$

$WHERE(X, f)$ filters all elements in X for which the function f does not evaluates to *true*. $WHERE(X, f)$ is only defined for the function f if the following condition holds:

$$T(X) \subseteq D(f) \wedge C(f) = \{true, false\}$$

$$TAKE(X, n) = \Big[x_1, x_2, ..., x_n\Big] \tag{5}$$

$TAKE(X, n)$ filters all but the first n elements of X. Thus $TAKE([1, 2, 3, 4, 5, 6], 4)$ would result in the sequence $[1, 2, 3, 4]$.

$$DROP(X, n) = \Big[x_{n+1}, x_{n+2}, ..., x_{n+i} \;\Big|\; n + i = |X|\Big] \tag{6}$$

$DROP(X, n)$ filters the first n elements from the set X. Thus $DROP([1, 2, 3, 4, 5, 6], 4)$ would result in the sequence $[5, 6]$.

All 3 filter operations maintain the type of the input sequence.

3.2.3 Set Operations

The basic set operations, slighly altered for use with sequences.

$$UNION(X, Y) = \Big[x_1, x_2, ..., x_n, y_1, y_2, ..., y_m \;\Big|\; n = |X| \wedge m = |Y|\Big] \tag{7}$$

Unlike the union of two sets, the union of 2 sequences is found simply by concatenating one to the other. This difference is due to the fact that for sequences, order is significant and duplicate

elements are allowed. $UNION(X, Y)$ thus represents the concatenation of X and Y. For example, $UNION([1, 2, 3], [3, 2, 1])$ results in the sequence $[1, 2, 3, 3, 2, 1]$.

$$INTERSECT(X, Y) = \left[x_i \;\middle|\; x_i = y_i \right] \tag{8}$$

$INTERSECTION(X, Y)$ is the sequence consisting of all elements of X where the same value is present in Y in the same position. For example, $INTERSECTION([1, 2, 3, 4, 5, 6], [4, 2, 2, 4])$ results in the sequence $[2, 4]$ since $x_2 = y_2$ and $x_4 = y_4$.

$$DIFFERENCE(X, Y) = \left[x_i \;\middle|\; x_i \neq y_i \right] \tag{9}$$

$DIFFERENCE(X, Y)$ is the sequence consisting of all elements of X where a different value is present in Y in the same position. For example, $DIFFERENCE([1, 2, 3, 4, 5, 6], [4, 2, 2, 4])$ results in the sequence $[1, 3, 5, 6]$.

All 3 set operations are defined only for sequences X and Y where $T(X) = T(Y)$ holds.

3.2.4 Order Operations

The order operations modify the order of the input sequence.

$$SORT(X) = \left[x_1, x_2, ..., x_n \;\middle|\; \forall_{1 \leq i < n}(x_i < x_{i+1}) \right] \tag{10}$$

$SORT(X)$ sorts X in ascending order. Thus $SORT([5, 2, 9, 1])$ results in the sequence $[1, 2, 5, 9]$.

$$REVERSE(X) = \left[x_n, x_{n-1}, ..., x_1 \;\middle|\; n = |X| \right] \tag{11}$$

$REVERSE(X)$ reverses X.

Both order operations maintain the type of the input sequence.

3.2.5 Boolean Reductions

The boolean reductions reduce their input to a single boolean value.

$$EQUAL(X, Y) = \begin{cases} true & if\ X = Y \\ false & otherwise \end{cases} \tag{12}$$

$EQUAL(X, Y)$ evaluates to *true* if X is identical to Y. $EQUAL(X, Y)$ is defined for all sequences X and Y where $T(X) \cap T(Y) \neq \emptyset$.

$$CONTAINS(X, x) = \begin{cases} true & if\ x \in X \\ false & otherwise \end{cases} \tag{13}$$

$CONTAINS(X, x)$ is *true* if X contains x.

$$ANY(X, f) = \begin{cases} true & if\ \exists_{x \in X}(f(x)) \\ false & otherwise \end{cases} \tag{14}$$

$$ALL(X, f) = \begin{cases} true & if\ \forall_{x \in X}(f(x)) \\ false & otherwise \end{cases} \tag{15}$$

$ANY(X, f)$ and $ALL(X, f)$ is *true* if $f(x)$ evaluates to *true* for any/all elements x in X. $ANY(X, f)$ and $ALL(X, f)$ are only defined for f if the following condition holds:

$$T(X) \subseteq D(f) \wedge C(f) = \{true, false\}$$

3.2.6 Numerical Reductions

The numerical reductions reduce their input to a single numerical value.

$$SIZE(X) = |X| \tag{16}$$

$SIZE(X)$ represents the number of elements in X. Note that $C(SIZE(X)) = \{0\} \cup \mathbb{Z}^+$ (The set of positive integers including zero) regardless of $T(X)$.

$$COUNT(X, y) = |[x \mid x \in X \wedge x = y]| \tag{17}$$

$COUNT(X, y)$ counts the number of elements in X which are equal to y. Note that just as with $SIZE$, $C(CONTAINS(X, y)) = \{0\} \cup \mathbb{Z}^+$ regardless of $T(X)$.

$$SUM(X) = \sum_{i=1}^{|X|} x_i \tag{18}$$

$SUM(X)$ calculates the sum of all elements in X. Unlike $SIZE$ and $CONTAINS$, $C(SUM(X))$ is entirely dependent upon the value of $T(X)$. If $T(X) = \mathbb{R}$ then $C(SUM(X)) = \mathbb{R}$.

$$MINIMUM(X) = x_1 \in SORT(X) \tag{19}$$
$$MAXIMUM(X) = x_n \in SORT(X) \text{ where } n = |X| \tag{20}$$

$MINIMUM(X)$ and $MAXIMUM(X)$ finds the smallest and largest element of X. Which element is to be considered the smallest or largest depends upon $SORT(X)$ which in turns depends upon the outcome of the simple comparisons between each element. Note that regardless of how $SORT(X)$ orders X, $C(MINIMUM(X)) = C(MAXIMUM(X)) = T(X)$.

$$AVG(X) = \frac{SUM(X)}{|X|} \tag{21}$$

$AVG(X)$ calculates the average value of the elements in X. Note that the division includes the fractional part, so $C(AVG(X)) = \mathbb{R}$. If $T(X)$ is not a set of numbers, the type and result of $AVG(X)$ depends entirely upon the type and result of $SUM(X)$.

$$FOLD(X, f, s) = f\Big(f\big(f\big(f(s, x_1), x_2\big), ...\big), x_n\Big) \tag{22}$$

$FOLD(X, f, s)$ performs a reduction of X by successively applying f to elements of X, starting with x_1 and s. The function f must be a binary function such that $(y, z) \in D(f)$ where $y \in C(f)$ and $z \in T(X)$.

3.2.7 Additional Operations

These additional operations do not fit within any specific category.

$$UNIQUE(X) = \Big[x_i \;\Big|\; \neg\exists_{j<i}\big(x_i = x_j\big)\Big] \tag{23}$$

$UNIQUE(X)$ removes all duplicate elements from X. Note that $T(UNIQUE(X)) = T(X)$.

$$ZIP(X, Y) = \Big[(x_1, y_1), (x_2, y_2), ..., (x_n, y_n) \;\Big|\; n = MIN\big(|X|, |Y|\big)\Big] \tag{24}$$

$ZIP(X, Y)$ zips elements from both sequences into a sequence of tuples. The type of $ZIP(X, Y)$ is the set of tuples of elements from X and Y:

$$T\big(ZIP(X, Y)\big) = \Big\{(x, y) \mid x \in X \wedge y \in Y\Big\}$$

$$MAP(X, f) = \Big[f(x) \;\Big|\; x \in X \Big] \tag{25}$$

$MAP(X, f)$ applies function f to each element in sequence X. The type of $MAP(X, f)$ depends on the function f:

$$T\big(MAP(X, f)\big) \subseteq C(f) \Leftrightarrow T(X) \subseteq D(f)$$

3.3 Model Transformations

Each of the operations from section 3.2 can be used to create composite functions, which we shall refer to as *transformations*. A transformation is a unary function which expects a sequence of some type as input. It consists of a chain of functions which we can define using the standard function composition syntax.

Definition 1:
If h and g are functions, $f = h \circ g$ denotes the composition of these functions such that given an input x, $f(x) = h\big(g(x)\big)$.

We may chain together an arbitrary number of functions in this manner, but it is clear that we must consider the arity, domain and codomain of each function in such a chain, if the transformation is to be well-defined.

All transformations in the model are unary; the input consists of a single sequence. As not all operations in the model are unary, this can lead to issues. Clearly, the output of $INTERSECT$ cannot be used as input for ALL, since the latter expects both a sequence and a function as input, while the former returns just a sequence.

To resolve this issue, we can use partial function application to reduce the arity of an operation. Since all operations in the model accept a sequence as the first argument, by fixing any arguments aside from the first, we can use the operations to compose a transformation as intended.

Definition 2:
We define the function $APPLY(f, A)$ to accept a function f and a sequence of arguments A. $APPLY(f, A)$ will then output a new unary function with each argument aside from the first fixed to the value of the elements of A, in order. Note that $APPLY(f, S) = f$ if $S = \emptyset$.

If f is a tertiary function mapping a sequence and two integers to some other type, $APPLY(f, 1, 2)$ will map to the unary function f' which expects just a sequence as input, while the second and third arguments have been fixed to the values 1 and 2. We can now define a composite transformation f using $APPLY$.

Definition 3:
If $h_1, h_2, ..., h_n$ are functions and $S_1, S_2, ..., S_n$ are sequences of input for these functions, excluding the first parameter, then we can use $APPLY$ to create a composition f of $h_1, h_2, ..., h_n$:

$$f = \big(APPLY(h_1, S_1)\big) \circ \big(APPLY(h2, S_2)\big) \circ ... \circ \big(APPLY(h_n, S_n)\big)$$

All other arguments, aside from the first, for each operation in f, will be fixed regardless of the input sequence passed to f. Even though each operation now requires the same number of arguments, we must still ensure that the input from one function lines up with the output of the previous function in the chain.

Definition 4:
If $h_1, h_2, ..., h_n$ are unary functions and $f : X \to Y$ is a composition of these functions

$$f = h_1 \circ h_2 \circ ... \circ h_n$$

then f is only well defined under the following condition:

$$\forall_{2 \leq i \leq n} \big(C(h_{i-1}) \subseteq D(h_i) \big)$$

By this definition, we find that $D(f) = D(h_1)$ and $C(f) = C(h_n)$, and thus the transformation is defined for all inputs $X \in D(h_1)$.

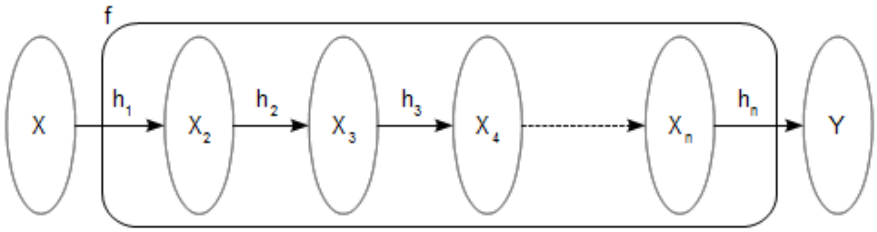


Figure 1: Overview of how the transformation f maps $X \to Y$.

Figure 1 shows how f is composed of a chain of functions. Note that the sets $X_2, X_3, ..., X_n$ can be completely disjoint from either X or Y, depending on the functions used to compose f. Indeed the implementation of the model considers these intermediate results to be implementation details and does not provide an interface to expose them.

3.4 Implementing the Model

A number of operations in the model (`map`, `fold`, `zip`) are staples of the functional programming universe. The implementation of these operations in a functional programming environment would be as easy as calling the respective functions with appropriate arguments. Similarly, partial function application and function composition is already supported in most functional programming environments in the form of function currying and higher-order functions. The type genericness of the model lends itself well to the polymorphic algebraic data types of functional languages such as those of the `ML` family.

As `C++` does not provide all of this functionality out of the box, a number of compromises has to be made when implementing the model. While `C++` does allow passing a function to another in the form of function pointers, this would be a poor substitute for higher-order functions since function pointers are generally not considered type-safe nor are they easy to use in a generic fashion.

Instead, function objects can be used. By implementing each operation as a function object, the functions can be passed to one another after instantiation. As each of these function objects needs to be type generic, they will have to be implemented as object templates, which can then be instantiated with any type. Using this approach, the composition of operations is a matter of storing a number of function objects, which can then be invoked in order when the transformation is performed. With the new variadic templates, this can be done with any number of operations in a type safe manner.

3.5 Applying the Model

Example 1:
Given these auxiliary definitions:

$$X = [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]$$
$$Y = RANGE(1, 7, 1) = [1, 2, 3, 4, 5, 6, 7]$$

$$f(x) = \begin{cases} true & if\ x\ mod\ 2 = 0 \\ false & otherwise \end{cases}$$
$$g(x) = x \cdot x$$

We may combine $WHERE$, MAP and ZIP in the following manner:

$$h_1 = APPLY\big(WHERE, SINGLETON(f)\big)$$
$$h_2 = APPLY\big(MAP, SINGLETON(g)\big)$$
$$h_3 = APPLY\big(ZIP, SINGLETON(Y)\big)$$
$$COMPOSITION = (h_3 \circ h_2 \circ h_1)$$

$$COMPOSITION(Z) = (h_3 \circ h_2 \circ h_1)(Z) = h_3\Big(h_2\big(h_1(Z)\big)\Big)$$
$$COMPOSITION(X) = \big[(0, 1), (4, 2), (64, 3), (1156, 4)\big]$$

The transformation $COMPOSITION$ defined in Example 1 accepts a sequence of integers, drops all odd elements of this sequence, squares the remaining elements and then zips this sequence with the sequence of integers from 1 to 7. Figure 2 shows an overview of the evaluation of $COMPOSITION(X)$ where X is the sequence of Fibonacci numbers below 100.

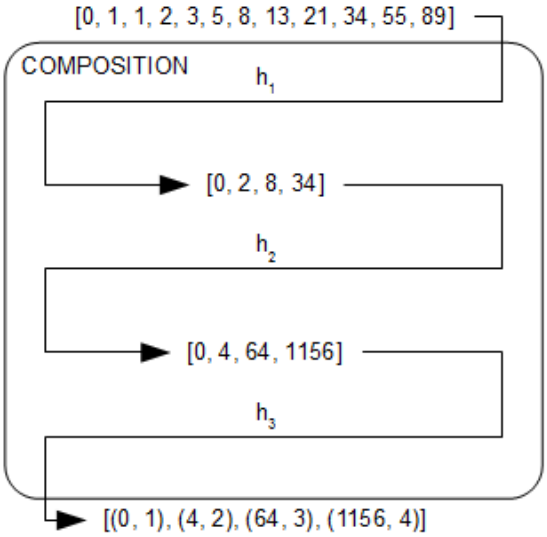


Figure 2: Overview of the evaluation of $COMPOSITION(X)$ from Example 1.

4 Model implementation

We will now present a fully-functional experimental implementation of list-comprehension using `C++11` variadic templates, based on the model introduced in section 3.

4.1 Overview - `stplib`

The Sequence Transformation Pipeline library (`stplib`) is implemented as a header-only library. Due to the way a `C++` compiler instantiates a class or function template, they have to be placed in a header file such that the instantiations are visible to the linker [5]. Since `stplib` makes heavy use of class templates, function templates, complete and partial template specializations, variadic function and class templates, template templates and various metaprogramming techniques, a header-only library was a feasible way to make the library usable. The header-only structure also makes the library easier to install and use, since no additional compiling and linking steps are necessary.

The `stplib` source code is published under the MIT License [6], which permits modification, extension and redistribution.
The library is available for download on its GitHub page `https://github.com/TueHaulund/stplib/`.

Due to the heavy use of templates and variadic templates, compiler support for `stplib` is still limited. The library has been succesfully compiled and tested on MinGW (4.8.1), GCC (4.8.1) and Clang (3.4.1). At the time of writing, the newest version of MSVC does not succesfully compile the library, due to limited support for variadic templates.

Included with the source code is a small test suite based on Boost.test [7]. The test suite also doubles as a program entry point and can thus be run as a standalone program. Instructions for building and running the test suite can be found in Appendix B. Note that only the test suite relies upon the Boost library being available, `stplib` itself does not make use of any third party library aside from the STL (the `C++` Standad Template Library).

The test suite and library can be built using the included SCons build-files. See the project GitHub page or Appendix A for build instructions.

The implementation of `stplib` consists of numerous header files, but a convenience header `stp.hpp` is included for ease-of-use. All `stplib` definitions and declarations are placed in the `stp` namespace. Note that `stp::detail` is reserved for the implementation, and any code within this namespace is subject to change.

The library is arranged into two distinct parts, just as the mathematical model. A range of predefined operations is included, as well as facilities for chaining these operations together as explained in section 3.3. The naming and grouping of these operations also match those of the model. One exception to this is the `union` operation, which has been renamed to `join` since `union` is a reserved keyword in `C++`.

4.2 Operations

All predefined operations in `stplib` consists of a struct placed in namespace `stp::detail` which overloads `operator()` and a function placed in namespace `stp` which constructs the operation without having to explicitly state the type of any arguments to the constructor.

Example 2:

```
1  namespace stp
2  {
3      namespace detail
4      {
5          template <typename Predicate>
6          struct any_type
7          {
8              any_type(const Predicate &pred) : pred_(pred) {}
9
10             template
11             <
12                 typename SequenceType,
13                 typename ValueType = typename SequenceType::value_type,
14                 typename PredType = typename std::result_of<Predicate(ValueType)>::type,
15                 typename = typename std::enable_if<std::is_convertible<PredType,
                       bool>::value>::type
16             >
17             bool operator()(const SequenceType &sequence) const
18             {
19                 return std::any_of(std::begin(sequence), std::end(sequence), pred_);
20             }
21
22             Predicate pred_;
23         };
24     }
25
26     template <typename Predicate>
27     detail::any_type<Predicate> any(const Predicate &pred)
28     {
29         return detail::any_type<Predicate>(pred);
30     }
31 }
```

Example 2 shows the implementation of the `any` operation defined in section 3.2.5. The operation will return *true* if the predicate returns *true* for one or more elements in the sequence. The function operator accepts only a single argument; the sequence which is being transformed. All auxiliary arguments (in this case there is only one, the predicate function) are passed to the object through the constructor.

As the composition of operations is achieved with the use of a variadic function template, the operations could not be implemented as actual functions. The reason for this is that `C++` does not support lazy evaluation, which means all function calls would be evaluated immediately, rather than being deferred until the input was passed to the transformation object. This is elaborated upon further in section 4.3.

By implementing each operation as a function object, the auxiliary parameters are stored in the operation as it is instantiated, but the actual evaluation does not take place until the transformation object calls the function operator with the input sequence. As the model uses partial-function evaluation to achieve this, having access to a functional programming environment with support for lazy evaluation and function currying would have made it possible to have the implementation follow the model far more accurately. In lieu of these features, function objects can be utilized to obtain a similar effect (See section 3.4).

Although each operation is implemented in a type-generic fashion, some preconditions do apply. For `any`, the result of calling the predicate with an element of the input sequence must be implicitly convertible to type `bool`. These restrictions are enforced with the use of additional template-parameters with default values which uses the `type_traits` library of the STL. This approach is

not generally necessary, since the template instantiations will generate compiler errors if a type does not meet the requirements of the operation. As these errors tend to be rather involved, these restrictions are provided to make sure the template instantiation fails as early as possible, thus reducing the complexity of any errors and increasing the usability of the library (See section 5.3.4).

Since the STL function `std::any_of` behaves exactly as `any` does in the model, the operation is implemented as a simply wrapper around `std::any_of`. Many of the operations in the library are implemented in this fashion.

Example 3:

```
1  namespace stp
2  {
3      namespace detail
4      {
5          struct to_map_type
6          {
7              template
8              <
9                  typename SequenceType,
10                 typename ValueType = typename SequenceType::value_type,
11                 typename FirstType = typename ValueType::first_type,
12                 typename SecondType = typename ValueType::second_type,
13                 typename PairType = typename std::pair<FirstType, SecondType>,
14                 typename MapType = typename std::map<FirstType, SecondType>
15             >
16             MapType operator()(const SequenceType &sequence) const
17             {
18                 MapType result;
19                 std::for_each(std::begin(sequence), std::end(sequence), [&](const
                       ValueType &i){result.insert(PairType(i.first, i.second));});
20                 return result;
21             }
22         };
23     }
24
25     detail::to_map_type to_map()
26     {
27         return detail::to_map_type();
28     }
29 }
```

Example 3 shows the implementation of the `to_map` operation. This operation is not present in the model, but was added to the library for the sake of usability. The library also contains a `to_list` and a `to_vector` operation, to provide compatibility with the most common STL containers.

The `to_map` operation does not take any arguments other than the input sequence, thus there are no template parameters for the compiler to infer. The helper function is still present for the sake of consistency. The operation uses `std::for_each` and an anonymous lambda function to generate a `std::map` from a sequence of 2-tuples. In this case, each tuple must define the types of the first and second value, as well as provide accessor functions for these values.

Any operation that requires a predicate or a function as input will accept either a function object or a lambda-function. Regular functions can not be stored by the operation, so to pass a function to an operation, the function call should be wrapped in a lambda or within a `std::function` from the <`functional`> header.

All operations in `stplib` defines `operator()` as a `const` member function. This ensures that the operation cannot alter its own internal state between executions, which in turn means that the evaluation of a transformation will always be deterministic and consistent.

4.3 Transformations

To facilitate the composition of operations, `stplib` uses a variadic class template called `stp_type` defined in the `stp::details` namespace. Since `C++` does not allow iteration over the elements of a template parameter pack, `stp_type` is defined as a recursive structure with a special base case defined as a partial template specialization of the general recursive case. This object is a function object, same as the operations that it is composed of. Calling the function operator with an input sequence will apply each transformation to the sequence in order.

If an `stp_type` with just 1 template argument is instantiated, the constructor will expect just a single operation which is stored as a private class member and then invoked on the input when `operator()` is called.

Example 4:

```
1  template <typename OpType>
2  class stp_type<OpType>
3  {
4      public:
5          template <typename SequenceType>
6          using return_type = typename std::result_of<OpType(SequenceType)>::type;
7
8          stp_type(OpType operation) : operation_(operation) {}
9
10         template <typename SequenceType>
11         return_type<SequenceType> operator()(SequenceType sequence) const
12         {
13             return operation_(sequence);
14         }
15
16     private:
17         OpType operation_;
18 };
```

Example 4 shows how `stp_type` is implemented for the base case, where the transformation consists of a single operation.

If `stp_type` is instantiated with n template arguments the recursive case will be used, as shown in Figure 3.

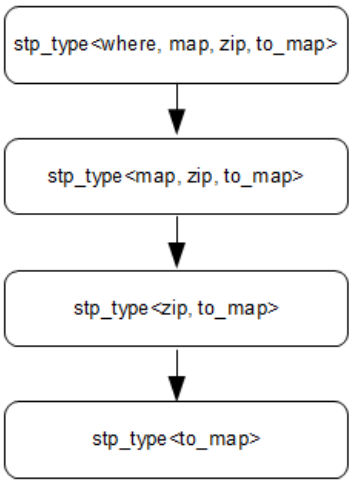


Figure 3: Inheritance diagram for `stp_type<where, map, zip, to_map>`.

The constructor will expect n arguments with the types of the arguments ordered in accordance with the template parameters. The first operation will be stored as a private member, and `stp_type` will inherit from itself with $n - 1$ parameters, which will in turn inherit from itself with $n - 2$ parameters, and so on until the operations have been stored across each of the base classes of the original `stp_type` object and the base case is reached which does not inherit from any class.

Example 5:

```
template
<
    typename OpType,
    typename ...Rest
>
class stp_type : public stp_type<Rest...>
{
    private:
        using base_type = stp_type<Rest...>;
        using this_type = stp_type<OpType, Rest...>;

        template <typename SequenceType>
        using op_return_type = typename std::result_of<OpType(SequenceType)>::type;

    public:
        template <typename SequenceType>
        using return_type = typename base_type::template
             return_type<op_return_type<SequenceType>>;

        stp_type(OpType operation, Rest... rest) : base_type(rest...),
             operation_(operation) {}

        template <typename SequenceType>
        return_type<SequenceType> operator()(SequenceType sequence) const
        {
            using base_return = typename
                 std::result_of<base_type(op_return_type<SequenceType>)>::type;
            using this_return = typename std::result_of<this_type(SequenceType)>::type;
            static_assert(std::is_same<base_return, this_return>::value, "Pipeline type
                 mismatch");

            return base_type::operator()(operation_(sequence));
        }

    private:
        OpType operation_;
};
```

Example 5 shows how `stp_type` is implemented for the recursive case. The return type of the function operator is evaluated at compile time, in a recursive manner much like that of the object instantiation.

Using `std::result_of`, the type of applying the first operation to the input sequence is determined. A templated `using` declaration is then used to query the base class for the type of applying the next operation to this new sequence type. This process continues until the base case is reached, whereby the recursion halts and the final result of applying the transformation is determined.

Each function operator in the inheritance hierarchy therefore has the same return type; the type of applying all operations in order to the given sequence. The type of the parameter for each function operator matches the type of the intermediate sequences, as illustrated in Figure 1.

As the template instantiation of `stp_type` takes place during compilation, any potential mismatches beetween operations or sequence types will be caught by the compiler. This compile time type-enforcement guarantees that no type errors will occur at run time, minimizing the risk of type-related bugs or faults.

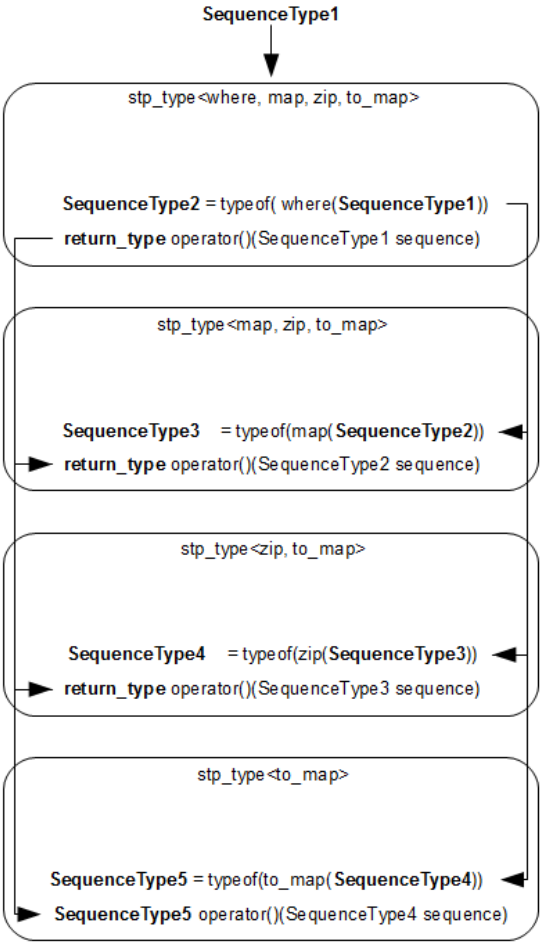


Figure 4: Illustration of the compile time evaluation of the return type of an `stp_type` object.

4.4 Interface

One of the advantages of using variadic templates in `stplib` is that the construction of a new transformation can be done with a simple function call, which will take care of the composition of operations. As the types involved in this process can become rather long-winded and complex, some simple helper functions are available to ensure that a user of the library will not have to deal with these types directly.

Example 6:

```
1 template
2 <
3     typename OpType,
4     typename ...Rest
5 >
6 detail::stp_type<OpType, Rest...> make_stp(OpType operation, Rest... rest)
7 {
8     return detail::stp_type<OpType, Rest...>(operation, rest...);
9 }
```

The `make_stp` function shown in example 6 is a simple variadic function template which will construct an `stp_type` object and return it. As `make_stp` is a function template, the compiler can infer the template arguments from the function arguments, which allows the user to create

an `stp_type` object without explicitly stating the template parameters. Combined with the `auto` specifier, which allows the compiler to infer the type of the variable based on the type of the value assigned initially, the user of the library will never need to explicitly refer to the type of an `stp_type` object.

Example 7:

```
1 detail::stp_type<detail::take_type,
2                  detail::drop_type,
3                  detail::count_type<std::string>> stp_obj(take(5),
4                                                           drop(2),
5                                                           count(std::string("xyz")));
6
7 auto stp_obj = make_stp( take(5), drop(2), count(std::string("xyz")) );
```

Example 7 illustrates how the use of `make_stp` can aid in the construction of an `stp_type` object. Clearly, explicitly stating the type of the transformation object is the vastly inferior approach in terms of length, readability and complexity.

The `make_stp` function also serves another purpose. Some operations, such as `where` or `sort_with` will expect a predicate function as a template parameter. If the user of the library wants to pass a lambda function to these operations, it will not be possible to refer to their type explicitly, as the type of a lambda is unique and determined only at compile time. Using `make_stp` and the `auto` specifier, the user will never need to express the type of the lambda directly in the code.

Note that the parameters to `make_stp` in the example above are actual functions, all of which return a fully constructed operation. Just as with `make_stp`, each operation in `stplib` can be constructed with a small helper function which allows the compiler to infer the template arguments, thus simplifying the interface for the user. Because `C++` uses eager evaluation, each of these functions are fully evaluated and replaced with the appropriate operation objects before `make_stp` is called.

Once the transformation object has been instantiated, the user can call the object like a function, supplying the input sequence and the transformation object will return the result:

Example 8:

```
1 std::vector<std::string> str_vec({"abc", "cba", "qwp", "xyz", "xyz", "xyz", "qwop"});
2 std::vector<std::string> str_vec2({"abc", "cba", "qwp", "xyz", "qwop", "qpl"});
3
4 auto stp_obj = make_stp( take(5), drop(2), count(std::string("xyz")) );
5
6 int xyz_count = stp_obj(str_vec); //xyz_count is two
7 xyz_count = stp_obj(str_vec2);     //xyz_count is one
```

Note how one transformation object can be reused with different input, without having to reconstruct the object from scratch.

5 Implementation Variations

In order to explore the advantages of the approach used for `stplib`, a few other list-comprehension libraries for `C++11` have been chosen for a comparison of the advantages and disadvantages of the respective implementation strategies.

As mentioned in section 2.1 the primary focus is the library interface and ease of use, but as the variadic template approach differs from the other implementation strategies in other areas as well, these will also be explored.

Arguably, ease of use is a highly subjective matter. In this context, ease of use does not refer to end user usability [8], but instead to API usability in terms of software developers [9].

The following implementations will be analyzed and compared to `stplib`:

1. `cpplinq` by Mårten Rånge [10]
2. `boolinq` by Anton Bukov [11]

The `cpplinq` library can be found at CodePlex: `http://cpplinq.codeplex.com/`.
The `boolinq` library can be found at GitHub: `https://github.com/k06a/boolinq`.

In the following sections, a brief analysis of the implementation strategies employed by both libraries will be presented.

5.1 `cpplinq`

`cpplinq` aims to recreate the functionality of `LINQ` for the Microsoft `.NET` platform in `C++`. The project has been in active development since 2012.

5.1.1 Implementation

In `cpplinq`, operations are chained together with the stream extraction operator `>>`. Each operation provides an overload for this operator, which applies the operation to the input and returns a new object which also overloads `>>`. Thus, to create a transformation, a chain of operations can be constructed, similar to the way the STL handles streaming of input or output.

Operations have been split into two groups; sequence aggregators and sequence operators.
Sequence aggregators aggregates a sequence into a sum, `std::vector`, `std::map` or something else (such as the `sum` operation in `stplib`), while the sequence operators transmutes a sequence into another sequence (such as the `where` or `map` operations in `stplib`).
The sequence aggregators and operators use sequence sources, which takes the concept of a collection such as arrays, STL containers or generated collections and makes it composable with the aggregator and sequence operators [12].

All sequence aggregators in `cpplinq` consists of two parts; a builder class, which implements the functionality, and a helper method which constructs an instance of the builder class.
All builder classes inherit from a `base_builder` class and therefore have certain criteria that needs to be met. The builder class needs to inherit from the base class, be copyable and movable, have a member `this_type` which decides the type of the builder class and a method `build` which accepts a sequence and returns the aggregated value.

The sequence operators in `cpplinq` consists of three parts; a sequence class, which implements the functionality, a builder class which along with the sequence class produces a new sequence, and a helper method which constructs an instance of the builder class.

Similarly to the sequence aggregators, the sequence operators inherits from a base class. This base class differs from the base class which the aggregators inherit from, as the operators need additional parameters, such as return type and references in order to return a sequence object.

A sequence operator is valid if it inherits from the base class (`base_range`), is copyable and movable, has a member type deciding the type of the sequence class, a member type deciding the type of values in the sequence, a member type deciding the return type and an enumerator which decides if the return type is a reference or not. Furthermore, it should have a method `front` which returns the current value as a return type, a method `next` which moves the sequence to the next value and a method which overloads the stream extraction operator. The overloading method accepts a builder class and returns the new sequence, thus making it the central component required for composing transformations.

5.1.2 Interface

Creating composite transformations with `cpplinq` is very similar to dealing with I/O streams with the STL. Transformations are constructed by applying the stream extraction operator to a series of operations in a manner similar to function call chaining. The following example transformation constructs a sorted vector, containing only even numbers from a predefined array.

Example 9:

```
1 using namespace cpplinq;
2
3 int ints[] = {3,1,4,1,5,9,2,6,5,4};
4 std::vector<int> sorted_numbers;
5
6 sorted_numbers = from_array(ints)
7              >>  where([](int i) {return i % 2 == 0;})    // Keep only even numbers
8              >>  orderby_ascending([](int i) {return i;}) // Sort ascending
9              >>  to_vector();                             // Create vector
```

The transformation in Example 9 uses four operations; `from_array, where, orderby_ascending` and `to_vector`. The composition is then simply declared with the desired order and lambda functions are passed to `where` and `orderby_ascending`.

As the input sequence is considered part of the transformation, the evaluation takes place as the transformation is defined. Subsequently, if a similar transformation is to be performed on a different input sequence, the transformation will have to be redefined with this new input sequence.

Alternatively, the transformation can be wrapped in a callable object or a function, with the input sequence as a parameter, such as in Example 10.

Example 10:

```
1 std::vector<info> transformation(const std::vector<info> &info_vec)
2 {
3     return from_vector(info_vec)
4         >> orderby_ascending([] (const info &i) {return i.last_name;})
5         >> thenby_ascending([] (const info &i) {return i.first_name;})
6         >> to_vector();
7 }
```

5.2 `boolinq`

The `boolinq` library is a `C++` header-only `LINQ` template library. Unlike `stplib` and `cpplinq`, `boolinq` provides support for `Qt` containers [13].

The following analysis will focus on the second iteration of the project, which was released April, 2013, but still receives regular updates. The first iteration of the project can be found on its homepage at Google Code: `https://code.google.com/p/boolinq/`.

5.2.1 Implementation

All operations in `boolinq` are defined as public member function templates of a single object, `LinqObj`. Each of these functions apply the appropriate operation to the sequence, and then returns a new `LinqObj`. Thus the composition of operations is a simple matter of invoking these member functions in order. This approach is sometimes called the *Named-Parameter Idiom* and is used as a way to supply a function or object with a variable amount of named arguments. In the case of `boolinq`, it serves to allow a variable number of operations to be applied to the sequence.

Aside from these member functions, a `LinqObj` also contains an instance of an enumerator class, which acts as a uniform wrapper around the different types of sequences. As an operation returns, the new `LinqObj` will be instantiated with an enumerator object which matches the type of the new sequence generated by the operation.

By implementing all operations in a single object, a large amount of work has to be done by the compiler when instantiating the object template for `LinqObj`. Code is generated for each operation, for each type of sequence used. As a single operation can alter the type of the sequence, the amount of member function template instantiations is in the order of $n \cdot m$ where n is the number of operations supported by the library, and m is the number of operations performed by the program.

Depending on the effectiveness of the compilers dead-code removal optimizations, not many of these instances will make it into the final binary as only m of them is actually used, but it will still have an impact on compilation times.

5.2.2 Interface

Performing a transformation in `boolinq` is done by creating an instance of the `LinqObj` object, and then calling the appropriate member functions in order. Since each member function returns a new `LinqObj`, the calls can be chained as in Example 11.

Example 11:

```
using namespace boolinq;

struct Person
{
    std::string name;
    int age;
};

Person src[] =
{
    {"Kevin",14},
    {"Anton",18},
    {"Agata",17},
    {"Terra",20},
    {"Layer",15},
};

std::vector<std::string> under_18;

under_18 = from(src)
          .where(  [](const Person &pers){return pers.age < 18;})
          .orderBy([](const Person &pers){return pers.age;})
          .select( [](const Person &pers){return pers.name;})
          .toVector();
```

Just as with `cpplinq`, transformations cannot be given a new input sequence unless wrapped in a function like in Example 10. Like with `stplib` it is possible to create and use transformations without ever explicitly stating the type of any of the operations or intermediate sequences, even without the use of `auto`.

5.3 Comparisons

In order to be able to reasonably judge the merits of the approach used to implement `stplib`, the following comparisons will be made to `cpplinq` and `boolinq`:

- Extensibility
- Binary size
- Source code size
- Source code complexity
- Type safety
- Overall interface usability

5.3.1 Library Extensibility

Aside from the predefined operations present in all three implementations, a user might require a custom operation. Preferably the library should be implemented in a way that facilitates this.

By using a variadic function template for the operation composition, `stplib` has full support for custom operations. Any function object or lambda can be used in a transformation, under the condition that the parameter and return types match those of the preceding and subsequent operations in the transformation. Therefore, `stplib` places no special requirements on the function objects used in a transformation.

With `cpplinq`, some conditions apply to function objects used as custom operations. Most notably, an operation object must overload the stream extraction operator and supply various type definitions used by the library when applying the operation. The full list of requirements is available in the documentation [12].

As `boolinq` implements each operation as a public member function, extensions to the library is only possible by modifying the `LinqObj` type directly in the library header. There is no mechanism for registering new operations with the library without modifying the source code.

5.3.2 Binary Size

As `stplib` uses a large amount of templates, which are all instantiated at compile time and thus does not significantly contribute to the size of the binary, a comparison of the size of binaries generated for each library is presented.

Three small test programs were constructed, each of which uses one of the three libraries. The test programs remove all even numbers from a `std::vector` containing the numbers from 1 to 10, the program then sorts the vector and prints the output.

All 3 binaries were built with the `GCC 4.8.1` compiler in a `unix` environment, using the `-Os` flag.

The test program written using `stplib`:

```cpp
#include <iostream>

#include "stp.hpp"

using namespace stp;

int main()
{
    std::vector<int> int_vec = std::vector<int>({7, 4, 6, 8, 5, 2, 9, 1, 3, 10});

    auto stp_obj = make_stp( where([](int i){return i % 2 == 1;}),
                             sort(),
                             to_vector() );

    std::vector<int> sorted_odds = stp_obj(int_vec);

    for(auto i : sorted_odds)
    {
        std::cout << i << std::endl;
    }
    return 0;
}
```

The test program written using `cpplinq`:

```cpp
#include <iostream>

#include "cpplinq.hpp"

using namespace cpplinq;

int main()
{
    std::vector<int> int_vec = std::vector<int>({7, 4, 6, 8, 5, 2, 9, 1, 3, 10});

    std::vector<int> sorted_odds = from(int_vec)
                                >> where([](int i){return i % 2 == 1;})
                                >> orderby_ascending([](int i){return i;})
                                >> to_vector();

    for(auto i : sorted_odds)
    {
        std::cout << i << std::endl;
    }
    return 0;
}
```

The test program written using `boolinq`:

```cpp
#include <iostream>

#include "boolinq.h"

using namespace boolinq;

int main()
{
    std::vector<int> int_vec = std::vector<int>({7, 4, 6, 8, 5, 2, 9, 1, 3, 10});

    std::vector<int> sorted_odds = from(int_vec)
                                  .where([](int i){return i % 2 == 1;})
                                  .orderBy([](int i){return i;})
                                  .toVector();

    for(auto i : sorted_odds)
    {
        std::cout << i << std::endl;
    }
    return 0;
}
```

As seen in table 1, the test programs implemented using `stplib` and `cpplinq` generated the smallest binaries, while the `boolinq` binary was more than three times larger than those of the other libraries.

Library used in test program	File size
`stplib`	15375 bytes
`cpplinq`	15827 bytes
`boolinq`	42927 bytes

Table 1: File sizes for each compiled test programming using the various libraries.

The three programs were disassembled using the `unix` tool `objdump`. In order to find the accumulated number of assembly instructions for each test program, the instructions were counted from the `<main>` label in the disassembled program, including all nested labels and functions called from `<main>`.

Test program library	Number of assembly instructions
`stplib`	684
`cpplinq`	729
`boolinq`	1945

Table 2: Accumulated number of assembly instructions across the three test programs.

Table 2 shows that the number of assembly instructions for each program somewhat follows the trend of the differences in file size. The `stplib` test program is roughly $2,9\%$ smaller than the `cpplinq` program and its assembly count is roughly 6.1% smaller than the instruction count of `cpplinq` as well. The assembly count of `boolinq` is roughly 3 times larger than the other libraries, which follows the trend of the file sizes as well.

5.3.3 Source Code Size

While not an absolute measure of complexity, SLOC (Source Lines of Code) can be a useful metric when estimating the complexity of each of the implementations. Since `stplib` is not as feature complete as `cpplinq` and `boolinq`, some alterations had to be made for a fair comparison to be possible.

Crucially, the point of measuring SLOC is to get a notion of the amount of code required for each general approach to solving the problem of operation composition. The comparison is thus only relevant in this context, and minor differences between each library could easily be construed as one implementation being more thorough with sanity checks or using a style of indentation that requires more vertical space.

By observing the amount of lines dedicated to so called *bookkeeping-code* (Commonly called *boilerplate-code*), the differences in complexity for each general approach will become more evident. All code which is not directly responsible for altering a sequence was considered *bookkeeping-code*.

Example 12:

```
#ifndef STP_JOIN_HPP
#define STP_JOIN_HPP

#include <iterator>
#include <type_traits>

namespace stp
{
    namespace detail
    {
        template
        <
            typename FixedSequenceType,
            typename FixedValueType = typename FixedSequenceType::value_type
        >
        struct join_type
```

```
17          {
18              join_type(const FixedSequenceType &sequence) : fixed_sequence_(sequence) {}
19
20              template
21              <
22                  typename SequenceType,
23                  typename ValueType = typename SequenceType::value_type,
24                  typename = typename std::enable_if<std::is_same<ValueType,
                        FixedValueType>::value>::type
25              >
26              SequenceType operator()(const SequenceType &sequence) const
27              {
28                  auto begin = std::begin(sequence);
29                  auto end = std::end(sequence);
30
31                  auto begin_fixed = std::begin(fixed_sequence_);
32                  auto end_fixed = std::end(fixed_sequence_);
33
34                  SequenceType result;
35                  result.reserve(std::distance(begin, end) + std::distance(begin_fixed,
                        end_fixed));
36                  result.insert(std::end(result), begin, end);
37                  result.insert(std::end(result), begin_fixed, end_fixed);
38                  return result;
39              }
40
41              FixedSequenceType fixed_sequence_;
42          };
43      }
44
45      template <typename SequenceType>
46      detail::join_type<SequenceType> join(const SequenceType &sequence)
47      {
48          return detail::join_type<SequenceType>(sequence);
49      }
50  }
51
52  #endif
```

In the code for `join` in Example 12, every line of code aside from lines 28 to 38 were considered *bookkeeping-code*. The purpose of this code is either to satisfy the verbosity requirements of the language (such as the header guards) or to facilitate the composition of operations when creating a new transformation.

To assist in the process of counting SLOC, excluding comments, the `SLOCCount` tool [14] was used. Note that SLOC in this context refers to physical lines of code, as opposed to logical lines of code.

Three equivalent implementations were made, by removing operations and functionality from `cpplinq` and `boolinq` until they could be considered identical to `stplib` in terms of feature-completeness.

Implementation	Total SLOC	Total SLOC in equivalent version
`stplib`	1177	1177
`cpplinq`	4397	3208
`boolinq`	636	379

Table 3: Total lines of code for all three implementations and their equivalent versions.

Implementation	Total lines of code	Total lines of bookkeeping code	Ratio
`stplib`	1177	966	0.22
`cpplinq`	4397	3063	0.44
`boolinq`	636	362	0.75

Table 4: Ratio between total lines of code and bookkeeping code in the three implementations.

The results in table 4 shows that `stplib` uses the least efficient implementation strategy in terms of the amount of code necessary to facilitate the composition of operations. For every line of code spent on bookkeping purposes, only 0.22 lines of code is spent performing each operation.

5.3.4 Source Code Complexity

While templates are necessary for implementing truly type-generic functions in `C++`, they add a lot of complexity to the codebase of all three libraries. As `stplib` relies the most on templates of the three, it is also the one that suffers the most from this added complexity. Even though the mathematical model defines the composition of operations in a relatively simple manner, a lot of considerations have to be ade when implementing this composition in a generic type-safe manner using variadic templates.

Despite the brevity of the implementation of the `stp_type` object, the source is neither easily readable nor easily understandable. Due to the complex nature of templates, any source code that makes heavy use of them suffers from a large amount of verbosity in order for the compiler to be able to discern the intention of the creator. By far the most used keyword in the `stplib` source code is `typename`, which tells the compiler that the subsequent expression is a qualified dependent type, something which would be impossible for the compiler to know without this otherwise extraneous keyword.

`C++` templates are notorious for these eccentricities, which serve only to clarify the meaning of the code for the compiler, but has the opposite effect for anyone else reading the source code. Another example of this is the keyword `template` which is mandatory before any nested name specifier in a qualified id [15], but which has a completely different meaning in all other contexts.

While none of these syntactic requirements needs to concern the users of `stplib`, the use of templates has another crucial drawback. If a user attempts to create a transformation where the types of the operations do not satisfy the requirements of the model, the compiler will generate a very lengthy and unintelligible error message, depending on where in the template instantiation the error occurs.

Generally these template errors are undecipherable, and will not help a user of the library pinpoint the problem. Some attempts have been made in `stplib` to utilize the new `static_assertion` functionality to make sure the template instantiation fails as early as possible with a meaningful error message, but it has not been possible to cover all possible errors.

The `cpplinq` and `boolinq` libraries also suffer from these drawbacks. However as the transformation composition in these libraries are handled with function call chaining, templates are only used to create type generic operations, while `stplib` employs templates for more involved metaprogramming purposes. Therefore the symptoms of these drawbacks are much more apparent in `stplib`.

5.3.5 Type Safety

As each of the three libraries are capable of performing complex transformations which might completely change the type of a sequence multiple times before a result is reached, type safety and correctness is a major concern.

All three implementations rely heavily on generic template programming, `stplib` most of all. One advantage of using templates, is that any potential errors or mismatches when a template is instantiated, will be caught at compile time and will thus not cause a run time error.

Example 13:

```
#include <iostream>
#include <vector>
#include <cstdint>

#include "stp.hpp"

using namespace stp;

int main()
{
    std::vector<std::int64_t> int64_vec;

    int64_vec.push_back(2147484644);
    int64_vec.push_back(2147484645);
    int64_vec.push_back(2147484646);
    int64_vec.push_back(2147484647);
    int64_vec.push_back(2147484648);

    std::vector<std::int32_t> int32_vec;

    int32_vec.push_back(1);
    int32_vec.push_back(2);
    int32_vec.push_back(3);
    int32_vec.push_back(4);
    int32_vec.push_back(5);

    auto transformed = make_stp( join(int64_vec) );

    for(auto i : transformed(int32_vec))
    {
        std::cout << i << std::endl;
    }

    return 0;
}
```

The program in Example 13 uses `stplib` to concatenate two vectors of different types. The code fails to compile since one of the preconditions of `join` is that the types of the two input sequences shall not be different from each other.

The same program implemented using either of the other two implementations will not result in a similar error. As `int64_t` is implicitly convertible to `int32_t`, no compilation error is encountered and an overflow occurs; the resulting sequence contains 5 negative numbers since 32 bits is not an adequate amount of space for the data in the 64 bit integers.

Each operation in `stplib` is itself responsible for ensuring that certain criteria are met before the operation can be performed. For the predefined operations in the library, this is accomplished using default template parameters and the `type_traits` library from the STL. By being strict about the types of the input, a run time error that could be difficult to track is avoided.

5.3.6 Interface Usability

The interfaces of all three libraries are based on that of the official `LINQ` interface and therefore shares many similarities with each other.

One of the main differences between `stplib`, `cpplinq` and `boolinq` is that the interface of `stplib` allows the user to define a transformation object which can be used with any number of different input sequences, while the other libraries fix a specific input sequence to the definition of the transformations.

In order to give a full assessment of the three interfaces, an API usability analysis, based on Steven Clarke's "Measuring API Usability" [9] will be presented.

The analysis revolves around a set of 12 different dimensions:

1. **Abstraction level**. The minimum and maximum levels of abstraction exposed by the API, and the minimum and maximum levels usable by a targeted developer.
2. **Learning style**. The learning requirements posed by the API, and the learning styles available to a targeted developer.
3. **Working framework**. The size of the conceptual chunk needed to work effectively.
4. **Work-step unit**. How much of a programming task must/can be completed in a single step.
5. **Progressive evaluation**. To what extend partially completed code can be executed to obtain feedback on code behavior.
6. **Premature commitment**. The amount of decisions the developers have to make when writing code for a given scenario and the consequences of those decisions.
7. **Penetrability**. How the API facilitates exploration, analysis, and understanding of its components, and how targeted developers go about retrieving what is needed.
8. **API elaboration**. The extent to which the API must be adapted to meet the needs of targeted developers
9. **API viscosity**. The barriers to change inherent in the API, and how much effort a targeted developer needs to expand to make a change.
10. **Consistency**. How much the rest of an API can be inferred once part of it is learned.
11. **Role expressiveness**. How apparent the relationship between each component exposed by an API and the program as a whole.
12. **Domain correspondence**. How clearly the API components map to the domain and any special tricks that the developer needs to be aware of to accomplish some functionality.

The set of 12 different dimensions used in an analysis of API usability, quoted from "Measuring API Usability" [9].

1. Due to the header-only nature of all three libraries, no implementation details are hidden from the user, but both `stplib` and `cpplinq` use nested namespaces to clarify the boundary between public interface and internal details. Any user which respects this boundary will only have access to the few interface functions used to construct operations and transformations and can therefore enjoy a relatively high level of abstraction.

2. Both `stplib` and `cpplinq` includes a full reference of all available operations, as well as a few usage examples and brief introductions. The documentation for `boolinq` is mainly based on example code. As the source code for all three libraries is very short, a targeted developer can explore the implementations themselves to gain insight (this is especially relevant in the case of `boolinq` which is less than 700 SLOC in total). The documentation for the official `LINQ` implementation is also freely available and might be marginally useful as all three libraries are very similar to this implementation in terms of the interface and feature-completeness.

3. The working framework of each of the libraries is very small, knowledge of only a few core interface functions and the operations that the user is applying will be needed to work effectively.

4. Declaring and applying a transformation is a single step process in both `cpplinq` and `boolinq`. For `stplib` the work-step unit is twice as large, since the creation of a transformation is a separate step from the actual application of the transformation.

5. Since all three libraries make heavy use of template metaprogramming, the feedback from the compiler, in case of an error, can be very intricate and undecipherable. Some effort has been put into supplying the user with useful feedback in the case of `stplib`, largely without luck. Neither `cpplinq` nor `boolinq` makes any attempts at addressing this concern.

6. In the case of `stplib` and `cpplinq` very little commitment is required. If at some point a design requires more flexibility from either of these libraries, they can be easily extended. In this regard, a developer using `boolinq` will not have as much flexibility.

7. As the size of the codebase of all three libraries is very small, any developer that needs to gain insight into the inner workings of one of them should easily be able to do so. On top of this, both `stplib` and especially `cpplinq` includes very detailed documentation of both interface and inner workings of the library.

8 + 9. API elaboration and viscosity was discussed in detail in section 5.3.1. Implementing new operations in each library requires rudimentary knowledge of `C++` templates.

10. The overall rate of consistency is very high for each library and allows a developer to infer most of the interface once a small part of it has been memorized.

11. Each library clearly documents the relationship between individual operations and the composite transformations. Each library has grouped different operations together with other related operations to make it easier for the developer to distinguish between them.

12. As each of the three libraries are an attempt to emulate the functionality of the official `LINQ` implementation, the domain correspondence should be a measure of how directly each operation can be mapped to an equivalent `LINQ` operation [3]. While `stplib` consciously avoids implementing each of these operations due to the context in which is was created, both the other libraries provide one-to-one mappings to the official `LINQ` implementation.

5.3.7 Comparison Summary

Library	Advantages/Disadvantages
`stplib`	+ Type errors caught at compile time + Relatively small source + Small binary size + Easily extensible + Full documentation available - Large amount of bookkeeping-code - Poor compiler support for variadic templates - Large amounts of template syntax verbosity
`cpplinq`	+ Small binary size + Relatively easily extensible + Full documentation available - Some type errors not caught by compiler - Very large source - Large amount of bookkeeping-code
`boolinq`	+ Very small source + Very small amount of bookkeeping-code - Long compilation time - Very large binary size - Not extensible - Some type errors not caught by compiler - Little to no documentation available

Table 5: Table showing the results of the comparisons of `stplib`, `cpplinq` and `boolinq`.

Overall, the three libraries each provide a straightforward yet elegant interface, which closely resembles that of the `LINQ` interface which the libraries are trying to replicate.

Although it is the simplest of the three, the approach used by `boolinq` has some crucial flaws which means the library is not at all extensible and suffers from a large amount of template instantiation bloat. The function chaining utilized by `cpplinq` is very similar to that used by `boolinq` yet retains full extensibility for the user.

6 Conclusion

With the formalized model of the list-comprehension concept; a fully functional implementation of this model was constructed and a brief analysis of all design and implementation choices made was presented. The variations between this implementation and the most popular existing implementations exposed the advantages and disadvantages inherent in the chosen implementation strategy.

The variadic template approach has proven to be a fully capable strategy for the implementation of a `LINQ`-like list-comprehension library for `C++`. The interface of `stplib` is completely generic and places no restrictions on the type of a valid operation, unlike the existing implementations which are only compatible with the predefined operations of the respective library or operations constructed specifically for that purpose.

On top of this, a variety of additional advantages were revealed throughout the construction of the library and this report:

- With the use of template metaprogramming, very strict preconditions can be enforced upon the types of the sequences being transformed which allows the compiler to detect type errors in the program during compilation.
- The source code is no longer than that of existing implementations, nor does it yield a larger binary.
- Due to the generic nature of the function composition, new operations can be constructed easily, even from existing functions using a minimal amount of code. This makes the implementation easily and fully extensible.

While these are significant advantages, some drawbacks to the large amount of template specific code in the implementation also became apparent:

- The strict type checking of input sequences are dependent upon the correct conditions being added as default template parameters or static assertions in the operation. This means that a large amount of code is necessary to direct the compiler to handle the type checking at compile time.
- The source code is plagued by the realities of template metaprogramming and an extreme amount of verbosity is required. This makes the source code longer and less succinct.
- The esoteric nature of template metaprogramming and especially of variadic templates means that compiler support is still very poor at the time of writing. This limits the usability of the implementation to a great extent.
- Any type errors will result in long-winded, unintelligible template instantiation errors, something for which `C++` templates are notorious. This complicates the development and debugging processes.

Since compiler support will undoubtedly improve drastically as time passes, some of the disadvantages of the approach might prove to be irrelevant in the future. Furthermore, the upcoming `C++1y` standard promises to introduce template concepts to the core-language, which will reduce the amount of code necessary to enforce any type restrictions in a rigid manner.

A fully optimized, feature-complete, production quality implementation of this variadic template approach would be a strong contender to the existing implementations.

7 References

[1] David N. Welton. *Programming Language Popularity*. 2013. URL: http://langpop.com/ (visited on 05/26/2014).

[2] Douglas Gregor, Jaakko Järvi, and Gary Powell. *Variadic Templates (Revision 3)*. 2006. URL: https://parasol.tamu.edu/~jarvi/papers/n2080.pdf (visited on 05/26/2014).

[3] Anders Hejlberg and Mads Torgersen. *The .NET Standard Query Operators*. 2007. URL: http://msdn.microsoft.com/en-us/library/bb394939.aspx (visited on 05/26/2014).

[4] Michael Sipser. *Introduction to the Theory of Computation*. Boston, USA: Cengage Learning, 2006. Chap. 0.2.

[5] Computer Science Department University of Maryland. *Compiling Template Classes*. 2003. URL: http://www.cs.umd.edu/class/fall2002/cmsc214/Projects/P2/proj2.temp.html (visited on 05/28/2014).

[6] Massachusetts Institute of Technology. *The MIT License*. URL: http://opensource.org/licenses/MIT (visited on 05/28/2014).

[7] Gennadiy Rozental. *Boost Test Library*. 2007. URL: http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html (visited on 05/28/2014).

[8] Microsoft Corporation. *Usability in Software Design*. 2000. URL: http://msdn.microsoft.com/en-us/library/ms997577.aspx (visited on 05/29/2014).

[9] Steven Clarke. "Measuring API Usability". In: *Dr. Dobb's Journal* Special Windows/.NET Supplement.May (2004), s6–s9. URL: http://itu.dk/stud/speciale/gamelib/APIUsability.pdf (visited on 05/29/2014).

[10] Mårten Rånge. *LINQ for C++*. 2014. URL: http://cpplinq.codeplex.com/ (visited on 05/29/2014).

[11] Anton Bukov. *Boolinq 2.0*. 2014. URL: https://github.com/k06a/boolinq (visited on 06/03/2014).

[12] Mårten Rånge. *Advanced usages of CppLinq*. 2014. URL: http://cpplinq.codeplex.com/wikipage?title=Advanced%20usages%20of%20CppLinq&referringTitle=Documentation (visited on 05/29/2014).

[13] Digia plc. *Qt Project*. 2014. URL: http://qt-project.org/ (visited on 06/03/2014).

[14] David A. Wheeler. *SLOCCount*. 2004. URL: http://www.dwheeler.com/sloccount/ (visited on 06/05/2014).

[15] *ISO C++ International Standard*. 2003. URL: http://cs.nyu.edu/courses/fall11/CSCI-GA.2110-003/documents/c++2003std.pdf (visited on 06/08/2014).

8 Appendices

A Documentation

Installation and build instructions are available on the project GitHub page: `https://github.com/TueHaulund/stplib`

Each operation in `stplib` is defined as a function object and a function template that constructs and returns the object. This allows the compiler to infer the types of the constructor arguments, rather than having them stated explicitly.

The following sections will list the function template signatures that construct each operation, as well as the requirements for each parameter and a small example. The 'sequence' refers to the input given to the transformation, which is passed to each operation. Any predicate can be either function objects or lambdas. To pass a regular function as an argument to an operation, use `std::function` from the `<functional>` header.

Boolean Reductions

all

```
1 template <typename Predicate>
2 detail::all_type<Predicate> all(const Predicate &pred)
```

`all` returns *true* if `pred` holds for all elements in the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- The result of calling `pred` on an element of type `SequenceType::value_type` must be implicitly convertible to type `bool`.

```
1 std::vector<int> int_vec({1, 2, 3, 4});
2 auto all_obj = all([](const int &i){return i % 2 == 0;});
3 bool result = all_obj(int_vec); //result = false
```

any

```
1 template <typename Predicate>
2 detail::any_type<Predicate> any(const Predicate &pred)
```

`any` returns *true* if `pred` holds for any element in the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- The result of calling `pred` on an element of type `SequenceType::value_type` must be implicitly convertible to type `bool`.

```
1 std::vector<int> int_vec({1, 2, 3, 4});
2 auto any_obj = any([](const int &i){return i % 2 == 0;});
3 bool result = any_obj(int_vec); //result = true
```

contains

```
template <typename ElementType>
detail::contains_type<ElementType> contains(const ElementType &val)
```

`contains` returns *true* if any element in the sequence is identical to `val`. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- `ElementType` must be implicitly convertible to `SequenceType::value_type`.
- `SequenceType::value_type` must define the equality operator.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto contains_obj = contains(3);
bool result = contains_obj(int_vec); //result = true
```

equal

```
template <typename SequenceType>
detail::equal_type<SequenceType> equal(const SequenceType &sequence)
```

`equal` returns *true* if both sequences are identical. The parameters must satisfy the following requirements:

- Both sequences must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- The `SequenceType::value_type` of both sequences must be identical.
- `SequenceType::value_type` must define the equality operator.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto equal_obj = equal(int_vec);
bool result = equal_obj(int_vec); //result = true
```

Filters

drop

```
detail::drop_type drop(const size_t &n)
```

`drop` removes the first `n` elements from the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::iterator`, `SequenceType.begin()`, `SequenceType.end()` and `SequenceType.erase()`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto drop_obj = drop(2);
bool result = drop_obj(int_vec); //result = {3, 4}
```

drop_while

```
template <typename Predicate>
detail::drop_while_type<Predicate> drop_while(const Predicate &pred)
```

`drop_while` removes elements from the sequence until `pred` returns *true* for an element. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType::iterator`, `SequenceType.begin()`, `SequenceType.end()` and `SequenceType.erase()`.
- The result of calling `pred` on an element of type `SequenceType::value_type` must be implicitly convertible to type `bool`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto drop_while_obj = drop([](const int &i){return i < 3;});
bool result = drop_while_obj(int_vec); //result = {3, 4}
```

take

```
detail::take_type take(const size_t &n)
```

`take` keeps the first `n` elements from the sequence, and removes the remaining elements. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::iterator`, `SequenceType.begin()`, `SequenceType.end()` and `SequenceType.erase()`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto take_obj = take(2);
bool result = take_obj(int_vec); //result = {1, 2}
```

take_while

```
template <typename Predicate>
detail::take_while_type<Predicate> take_while(const Predicate &pred)
```

`take_while` keeps elements from the sequence until `pred` returns *true* for an element, it then removes the remaining elements. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType::iterator`, `SequenceType.begin()`, `SequenceType.end()` and `SequenceType.erase()`.
- The result of calling `pred` on an element of type `SequenceType::value_type` must be implicitly convertible to type `bool`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto take_while_obj = take_while([](const int &i){return i < 3;});
bool result = take_while_obj(int_vec); //result = {1, 2}
```

where

```
template <typename Predicate>
detail::where_type<Predicate> where(const Predicate &pred)
```

`where` removes all elements from the sequence for which `pred` does not return *true*. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- The result of calling `pred` on an element of type `SequenceType::value_type` must be implicitly convertible to type `bool`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto where_obj = where([](const int &i){return i % 2 == 0;});
bool result = where_obj(int_vec); //result = {2, 4}
```

Miscellaneous

map

```
template <typename UnaryOperation>
detail::map_type<UnaryOperation> map(const UnaryOperation &unop)
```

`map` calls `unop` on each element of the sequence, and returns a new sequence composed of the resulting values. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.

The resulting sequence will be of type `std::vector<OpType>` where `OpType` is the type returned by calling `unop` with `SequenceType::value_type` as the parameter.

```
std::vector<int> int_vec({1, 2, 3});
auto map_obj = map([](const int &i){return std::string(" ", i);});
std::vector<std::string> result = map_obj(int_vec); //result = {" ", "  ", "   "}
```

to_list

```
detail::to_list_type to_list()
```

`to_list` returns a `std::list` containing the elements of the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.

The resulting sequence will be of type `std::list<SequenceType::value_type>`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto to_list_obj = to_list();
std::list<int> result = to_list_obj(int_vec); //result = {1, 2, 3, 4}
```

to_map

```
detail::to_map_type to_map()
```

`to_map` returns a `std::map` containing the elements of the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must define `value_type::first_type` and `value_type::second_type` (such as `std::pair` does).

The resulting sequence will be of type `std::map<SequenceType::value_type::first_type, SequenceType::value_type::second_type>`.

```
std::vector<std::pair<std::string, int>> pair_vec({
    std::pair<std::string, int>("one",   1),
    std::pair<std::string, int>("two",   2),
    std::pair<std::string, int>("three", 3)});

auto to_map_obj = to_map();
std::map<std::string, int> result = to_map_obj(pair_vec);
//result = {"one" : 1, "two" : 2, "three" : 3}
```

to_vector

```
detail::to_vector_type to_vector()
```

`to_vector` returns a `std::vector` containing the elements of the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.

The resulting sequence will be of type `std::vector<SequenceType::value_type>`.

```
std::list<int> int_list({1, 2, 3, 4});
auto to_vector_obj = to_vector();
std::vector<int> result = to_vector_obj(int_list); //result = {1, 2, 3, 4}
```

unique

```
detail::unique_type unique()
```

`unique` removes all duplicate elements from the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()`, `SequenceType.end()` and `SequenceType.push_back()`.
- `SequenceType::value_type` must define the equality operator.

```
std::vector<int> int_vec({1, 2, 2, 1});
auto unique_obj = unique();
std::vector<int> result = unique_obj(int_vec); //result = {1, 2}
```

zip

```
template <typename SequenceType>
detail::zip_type<SequenceType> zip(const SequenceType &sequence)
```

`zip` combines the two sequences into a single sequence of `std::pair`. The parameters must satisfy the following requirements:

- Both sequences must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.

The resulting sequence will be of type `std::vector<std::pair<value_type1, value_type2>>`. As zip uses `std::pair`, it is compatible with `to_map`.

```
std::vector<int> int_vec({1, 2, 3, 4});
std::vector<std::string> string_vec({"one", "two", "three", "four"});

auto zip_obj = zip(int_vec);
std::vector<std::pair<>> result = zip_obj(string_vec);
//result = {"one" : 1, "two" : 2, "three" : 3, "four" : 4}
```

Numerical Reductions

avg

```
detail::avg_type avg()
```

`avg` returns the average of the elements of the sequence as a double. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must be default-constructible.
- `SequenceType::value_type` must define the addition operator.

If the sequence is empty, `avg` will throw `std::range_error`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto avg_obj = avg();
double result = avg_obj(int_vec); //result = 2.5
```

count

```
template <typename ElementType>
detail::count_type<ElementType> count(const ElementType &val)
```

`count` returns the number of occurences of an element identical to `val` in the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()`, `SequenceType.end()` and `SequenceType::iterator`.
- `ElementType` must be implicitly convertible to `SequenceType::value_type`.

```
std::vector<int> int_vec({1, 2, 2, 4});
auto count_obj = count(2);
unsigned int result = count_obj(int_vec); //result = 2
```

fold

```
template
<
    typename BinaryOperation,
    typename InitType
>
detail::fold_type<BinaryOperation, InitType> fold(const BinaryOperation &binop, const
     InitType &init)
```

`fold` performs an accumulation of the elements of the sequence, using `binop` starting with `init` and the first element of the list. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- The result of `binop(InitType, SequenceType::value_type)` must be implicitly convertible to `InitType`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto fold_obj = fold(std::plus<int>(), 5);
int result = fold_obj(int_vec); //result = 15
```

max

```
detail::max_type max()
```

`max` returns the maximum element of the sequence, as defined by the less-than operator. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must define the less-than operator.

If the sequence is empty, `max` will throw `std::range_error`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto max_obj = max();
int result = max_obj(int_vec); //result = 4
```

min

```
detail::min_type min()
```

`min` returns the minimum element of the sequence, as defined by the less-than operator. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must define the less-than operator.

If the sequence is empty, `min` will throw `std::range_error`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto min_obj = min();
int result = min_obj(int_vec); //result = 1
```

size

```
detail::size_type size()
```

`size` returns the amount of elements in the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::iterator`, `SequenceType.begin()` and `SequenceType.end()`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto size_obj = size();
int result = size_obj(int_vec); //result = 4
```

sum

```
detail::sum_type sum()
```

`sum` returns the sum of the elements in the sequence as defined by the addition operator. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must be default-constructible.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto sum_obj = sum();
int result = sum_obj(int_vec); //result = 10
```

Order Operations

reverse

```
detail::reverse_type reverse()
```

`reverse` reverses the order of elements in the sequence. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType.begin()` and `SequenceType.end()`.

```
std::vector<int> int_vec({1, 2, 3, 4});
auto reverse_obj = reverse();
std::vector<int> result = reverse_obj(int_vec); //result = {4, 3, 2, 1}
```

sort

```
detail::sort_type sort()
```

`sort` sorts the sequence according to the less-than operator. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType.begin()` and `SequenceType.end()`.

```
std::vector<int> int_vec({4, 2, 1, 3});
auto sort_obj = sort();
std::vector<int> result = sort_obj(int_vec); //result = {1, 2, 3, 4}
```

sort_with

```
template <typename Predicate>
detail::sort_with_type<Predicate> sort_with(const Predicate &pred)
```

`sort_with` sorts the sequence according to `pred`. The parameters must satisfy the following requirements:

- The sequence must define `SequenceType::value_type`, `SequenceType.begin()` and `SequenceType.end()`.
- The result of `pred(SequenceType::value_type, SequenceType::value_type)` must be implicitly convertible to type `bool`.

```
std::vector<int> int_vec({4, 2, 1, 3});
auto sort_with_obj = sort_with([](const int &i, const int &j){return i < j;});
std::vector<int> result = sort_with_obj(int_vec); //result = {1, 2, 3, 4}
```

Set Operations

difference

```
template <typename SequenceType>
detail::difference_type<SequenceType> difference(const SequenceType &sequence)
```

`difference` returns a sequence consisting of all elements that are present in the first sequence while not being present at the same position of the second sequence. The parameters must satisfy the following requirements:

- Both sequences must define `SequenceType::value_type`, `SequenceType.push_back()`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must be the same for both sequences.

```
std::vector<int> int_vec({1, 2, 3, 4});
std::vector<int> other_int_vec({0, 2, 3, 5});
auto difference_obj = difference(other_int_vec);
std::vector<int> result = difference_obj(int_vec); //result = {0, 5}
```

intersect

```
template <typename SequenceType>
detail::intersect_type<SequenceType> intersect(const SequenceType &sequence)
```

`intersect` returns a sequence consisting of all elements that are present in the first sequence and also present in the second sequence in the same position. The parameters must satisfy the following requirements:

- Both sequences must define `SequenceType::value_type`, `SequenceType.push_back()`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must be the same for both sequences.

```
std::vector<int> int_vec({1, 2, 3, 4});
std::vector<int> other_int_vec({0, 2, 3, 5});
auto intersect_obj = intersect(other_int_vec);
std::vector<int> result = intersect_obj(int_vec); //result = {2, 3}
```

join

```
template <typename SequenceType>
detail::join_type<SequenceType> join(const SequenceType &sequence)
```

`join` returns the concatenation of the two sequences. The parameters must satisfy the following requirements:

- Both sequences must define `SequenceType::value_type`, `SequenceType.reserve()`, `SequenceType.insert()`, `SequenceType.begin()` and `SequenceType.end()`.
- `SequenceType::value_type` must be the same for both sequences.

```
std::vector<int> int_vec({1, 2, 3, 4});
std::vector<int> other_int_vec({4, 4});
auto join_obj = join(other_int_vec);
std::vector<int> result = join_obj(int_vec); //result = {1, 2, 3, 4, 4, 4}
```

Generators

Generators should not be part of a transformation pipeline, but can be used to generate sequences as input for any of the operations.

range

```
template
<
    typename IntervalType,
    typename StepType,
    typename RangeType = std::vector<IntervalType>
>
RangeType range(const IntervalType &start, const IntervalType &end, const StepType
    &step)
```

`range` will generate a sequence of elements, ranging from `start` to `end`, with each element being `step` larger than the previous element. `step` can be omitted in which case it will be fixed to 1. The parameters must satisfy the following requirements:

- Both `IntervalType` and `StepType` must be scalar types.

The resulting sequence will be of type `std::vector<IntervalType>`.

```
std::vector<int> result = range(1, 10, 3); //result = {1, 4, 7}
```

repeat

```
template
<
    typename ValueType,
    typename RepeatType = typename std::vector<ValueType>
>
RepeatType repeat(const ValueType &val, size_t n)
```

`repeat` will generate a sequence of `n` elements identical to `val`.

The resulting sequence will be of type `std::vector<ValueType>`.

```
std::vector<int> result = repeat(1, 3); //result = {1, 1, 1}
```

B Testing

A test suite is included with `stplib`. The test program acts as a program entry point and performs a large number of tests of the predefined operations and various transformation objects.

An SConstruct setup is included, which will build and run the tests on a variety of platforms. The SCons script will attempt to guess which compiler to use, but if a specific compiler is to be used it must be included as a command-line parameter when invoking SCons.

The test suite uses Boost.test, thus making Boost required to build and run the tests. Once built, the test suite can be invoked with a number of different command-line parameters. For a full list of parameters, refer to the Boost.test documentation (`http://www.boost.org/doc/libs/1_44_0/libs/test/doc/html/utf/user-guide/runtime-config/reference.html`).

Aside from Boost, the test suite uses some parts of the STL. The test suite can be found in the main `stplib` repository, in the *test* folder.

The tests use a text fixture consisting of several different test sequences of different types. A few auxiliary functions which can aid the tests in comparing floating-point values are also present. For each section of the test suite, an instance of the fixture is constructed and made available to the tests. Some `stplib` objects are declared and fed test sequences from the fixture. Finally, the results are compared to the expected result with a number of Boost.test assertion macros.

Example 14:

```
1 BOOST_CHECK( drop_5(ordered_ints) == std::vector<int>({6, 7, 8, 9, 10}) );
2 BOOST_CHECK( compare_range(range(-1.0f, 1.0f), std::vector<float>({-1.0f, 0.0f})) );
3 BOOST_CHECK_THROW( min_obj(empty_int_vec), std::range_error );
```

The test suite consists of the following sections:

- Operation tests - each operation is instantiated with different arguments and tested outside a transformation.
- Level 1 transformation tests - A range of different transformations consisting of just a single operation.
- Level 2 transformation tests - Several transformations consisting of two operations.
- Level 3 transformation tests - A few exotic transformations consisting of three or more operations.